Program Semantics and mechanized proof

PROGRAM SEMANTICS AND MECHANIZED PROOF by R. MILNER

MATHEMATICAL CENTRE TRACTS 82, 1976, 3-44

PROGRAM SEMANTICS AND MECHANIZED PROOF

R. MILNER
University of Edinburgh, Edinburgh, U.K.

1. INTRODUCTION

In the last seven or eight years strong advances have been made in the mathematical description of the meaning of programming languages. Before this, the semantic description (in contrast to the syntactic description, which was quite formal and was often given greater weight) was presented rather informally, and inevitably contained ambiguities and left out details, and the result was that the same language acquired different meanings in different implementations.

The work of Strachey and Scott and their followers has brought about an enormous improvement. Strachey was dissatisfied both with the informality of the existing language descriptions and with their dependence on the notion of evaluation, and when Scott provided the mathematical models which he was looking for it was a rather short time before the whole of then-existing languages, such as ALGOL 60, could be mathematically defined in an elegant manner. And because the description is mathematical, it is now possible both to study concepts underlying programming languages and to conduct proofs concerning (for example) the equivalence of different constructs in one language, or of constructs in different languages.

The current literature contains quite a few examples of mathematical descriptions of languages, but it is less easy to find reports of proofs about languages. This is perhaps because proofs about real languages tend to be long and difficult both to present and to read. The aim of this paper is to remedy this deficiency to some extent. We take a very simple language, which admittedly illustrates only some of the techniques which have been developed for language description, and in the next section we study its operational semantics (semantics by abstract machine, or by evaluation).

In section 3, using that small part of Scott's work on semantic domains which is reported in my paper "Models of LCF", we present its denotational semantics in the style originated by Strachey, and show that the two semantic descriptions are indeed equivalent in an appropriate sense.

Section 4 gives an alternative semantics for the language, using the technique of continuations; it is also demonstrated that this technique can more naturally handle certain extensions to the language (in particular the introduction of error exits, and other features which allow the "normal" flow of control to be diverted). We again give the proof that - for the non-extended language - the new semantics is equivalent to the old.

Finally in section 5, using as a basis the formal deductive calculus described in the second half of "Models of LCF", we discuss the problem of mechanizing the proof of section 4. The emphasis throughout the paper is on the detail of the proofs, since we wish to convince the reader as far as possible, by leaving as few gaps as possible, that the proof strategy of section 5 will actually work. I hope that the reader will also be encouraged to believe that proofs about larger languages will indeed be amenable to similar strategies; it is an unfortunate fact that proofs about programs and languages are on the whole so long and tedious in comparison to their intellectual content that no human being is likely to have the patience to convince himself (even less, to convince others) that they are *correct* proofs.

Not many references to the literature are given in the main part of the paper; instead, I have discussed some of the relevant papers in the final section.

2. OPERATIONAL SEMANTICS

2.1. Discussion

We will consider throughout these lectures a simple programming language L which is well-understood by everyone, and indeed possesses very few features of interest. May aim is to consider *styles* of proof about languages rather than sophisticated language "features", since I believe that these styles are also appropriate to more complex languages. It will be apparent that even for such a simple language as L the detailed proofs are nor particularly

easy to read; this is not so much because of their length as because of the high ratio of technical manipulation to real mathematical content. To put it more crudely, the proofs are tedious. It has been remarked more than once that no correctness proof of a program provides greater certainty of the program's correctness than does thorough "debugging", unless the proof is mechanically checked, and this is equally true of proofs about programming languages. If we first examine the mechanizability of some proofs about a simple language, we hope to reach a position from which we can advance to proofs about more complex languages.

In this section we introduce L and describe it operationally - that is, using an abstract machine and its state transitions. The technique derives from Landin [10] and is essentially the basis of the method used to define PL/1 [11,12]. We then describe an alternative operational model, using a method learnt from Plotkin and employed by him in [13] for various λ-calculi. The purpose of this second model is as an intermediary between the abstract machine model and the denotational semantic description to be studied in following sections.

Operational and denotational semantics play complementary roles, and I believe both will continue to be necessary. An operational definition gives a guide to implementation, and as such it is likely to be in the language designer's mind more or less explicitly from the outset, since he is always aiming at a language which admits an efficient implementation. On the other hand, to describe a language by giving an abstract denotation for each phrase is at least a guard against redundancy and "adhocness"; more importantly, the language is thus defined independently of the structure of an abstract machine (which however abstract, inevitably contains some arbitrary structure), and the denotational definition is more succinct - often by a factor of three or more in length. And perhaps most importantly, the denotational definition admits *proofs* about the language, which are either impossible or very cumbersome in terms of its operational definition.

2.2. The language L

Constants : 0,1,2,...,true,false
Variables : $x_0, x_1, \ldots$
Integer operations: +,-,×,...
Boolean operations: =,>,<,...

We use c,x,b,e,p,iop,bop to range over respectively constants, variables, boolean expressions, integer expressions, programs, integer operations, boolean operations.

Expressions : e ::= x|n|e_1 iop e_2

Boolean expressions: b ::= e_1 bop e_2|true|false

Programs : p ::= null|x:=e|p_1;p_2|if b then p_1 else p_2|
while b do p_1

(We have omitted the use of parentheses; we are concerned not with parsing but with the phrase structure which results from parsing).

These constitute the three types of phrase in language L.

2.3. The S,M,C machine

A machine state is a triple <S,M,C>, where

- the value stack S ∈ (Phrases)*
- the memory M ∈ (Constants)∞
- the control stack C ∈ (Phrases ∪ Operations ∪ {if,assign,while})*.

The *value stack* holds results (i.e. constants) and is also used to keep phrases whose execution is deferred.

The *memory* M = $m_0,m_1,\ldots$ holds current values of the variables $x_0,x_1,\ldots$.

The *control stack* holds phrases and operations awaiting execution.

The *Transition rules* of the machine are as follows, given by a relation ⇒ over states (in these rules " · " prefixes an element to a stack, and M[n/i] means $m_0,\ldots,m_{i-1},n,m_{i+1},\ldots$).

I⇒ <S,M,c·C> ⇒ <c·S,M,C>.

<S,M,x_i·C> ⇒ <m_i·S,M,C> .

II⇒ <n_2·n_1·S,M,+·C> ⇒ <n_1+n_2·S,M,C>

... etc. for all iops.

III⇒ <n_2·n_1·S,M,=·C> ⇒ <t·S,M,C> where t is true if n_1 = n_2,
false otherwise

... etc. for all bops.

IV⇒ <S,M,null·C> ⇒ <S,M,C>

<S,M,(x_i:=e)·C> ⇒ <i·S,M,e·assign·C>

<S,M,(p_1;p_2)·C> ⇒ <S,M,p_1·p_2·C>

$\langle S,M,(\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)\cdot C\rangle \Rightarrow \langle p_2\cdot p_1\cdot S,M,b\cdot\underline{\text{if}}\cdot C\rangle$

$\langle S,M,(\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1)\cdot C\rangle \Rightarrow \langle p_1\cdot b\cdot S,M,b\cdot\underline{\text{while}}\cdot C\rangle$.

V⇒ $\langle \underline{n}\cdot\underline{i}\cdot S,M,\underline{\text{assign}}\cdot C\rangle \Rightarrow \langle S,M[\underline{n}/i],C\rangle$

$\langle \underline{\text{true}}\cdot p_2\cdot p_1\cdot S,M,\underline{\text{if}}\cdot C\rangle \Rightarrow \langle S,M,p_1\cdot C\rangle$

$\langle \underline{\text{false}}\cdot\ldots\ldots\ldots\ldots\rangle \quad \langle\ldots\ldots p_2\ldots\rangle$

$\langle \underline{\text{true}}\cdot p_1\cdot b\cdot S,M,\underline{\text{while}}\cdot C\rangle \Rightarrow \langle S,M,(p_1;\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1)\cdot C\rangle$

$\langle \underline{\text{false}}\cdot\ldots\ldots\ldots\ldots\ldots\rangle \Rightarrow \langle S,M,\underline{\text{null}}\cdot C\rangle$.

Note that $\Rightarrow$ is deterministic. We use $\overset{n}{\Rightarrow}$ to denote the n-th power of the relation $\Rightarrow$, and $\overset{*}{\Rightarrow}$ for its transitive reflexive closure.

We define EVAL: Programs × Memories → Memories by:

$$\text{EVAL}(p,M) = M' \quad \text{iff} \quad \langle \varepsilon,M,p\cdot\varepsilon\rangle \overset{*}{\Rightarrow} \langle \varepsilon,M',\varepsilon\rangle\ ,$$

where ε stands for an empty stack. Notice that EVAL is a partial function.

2.4. The reduction relation →

The S,M,C machine is abstract, but rather arbitrary in the method chosen to control evaluation - that is, one might have used other structures in preference to a pair of stacks and a memory vector. Some of this arbitrariness may be removed by axiomatizing a reduction relation → over Phrases × Memories. We give the axioms and rules of a simple formal deductive system.

I→ $x_i,M \rightarrow \underline{m}_i,M$.

II→ $(e_1\ \text{op}\ e_2),M \rightarrow (e_1'\ \text{op}\ e_2),M$ if $e_1,M \rightarrow e_1',M$

$(\underline{n}\ \text{op}\ e),M \rightarrow (\underline{n}\ \text{op}\ e'),M$ if $e,M \rightarrow e',M$

$(\underline{n}_1+\underline{n}_2),M \rightarrow \underline{n_1+n_2},M$

... etc, for all iops.

$(\underline{n}_1=\underline{n}_2),M \rightarrow \underline{\text{true}},M$ / $\rightarrow \underline{\text{false}},M$ } according as $n_1 = n_2$ or not,

... etc, for all bops.

III→ $(x_i:=e),M \rightarrow \underline{\text{null}},M[\underline{n}/i]$ if $e,M \overset{*}{\rightarrow} \underline{n},M$.

IV→ $(p_1;p_2),M \rightarrow (p_1';p_2),M'$ if $p_1,M \rightarrow p',M'$

$(\underline{\text{null}};p),M \rightarrow p,M$.

V→ $(\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2),M \to p_1,M$ if $b,M \overset{*}{\to} \underline{\text{true}},M$

..... $\to p_2,M$ if $b,M \overset{*}{\to} \underline{\text{false}},M$.

VI→ $(\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1),M \to (p_1;\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1),M$ if $b,M \overset{*}{\to} \underline{\text{true}},M$

..... $\to \underline{\text{null}},M$ $\underline{\text{if}}\ b,M \overset{*}{\to} \underline{\text{false}},M$

REMARK. To be fully formal, we should be clear that the sentences of this formal system are of the form

$$\phi,M \to \phi',M' ,$$

where ϕ,ϕ' are program phrases; $\phi,M \overset{*}{\to} \phi',M'$ is not a sentence. Thus rule III→ has not just a single hypothesis $e,M \overset{*}{\to} \underline{n},M$, but k hypotheses (for some $k \geq 0$) of the form

$$e_i,M \to e_{i+1},M \qquad (0 \leq i < k),$$

where $e_i = e$, $e_k = \underline{n}$. The same remark holds for V→ and VI→.

We sketch the proof that → is deterministic. First, one shows that for each e,M there is at most one pair e',M' such that

$$e,M \to e',M'$$

and that M' = M. It follows that in the case of $\overset{*}{\to}$, for each e,M there is at most one pair $\underline{n},M'$ such that

$$e,M \overset{*}{\to} \underline{n},M'$$

and that M' = M. Similar results hold for boolean expressions. These proofs proceed by induction on the structure of expressions. Analogously, one then shows that if $p,M \to p',M'$ then p',M' is unique.

It follows that the partial function

$$\text{eval: Programs} \times \text{Memories} \to \text{Memories}$$

is well-defined as follows:

$$\mathrm{eval}(p,M) = M' \text{ iff } p,M \xrightarrow{*} \underline{\mathrm{null}},M' .$$

2.5. EVAL = eval

One might hope to prove the equivalence of EVAL and eval by induction on the structure of programs. This fails just because of the while construct, and we have to resort to induction on the length of computation.

LEMMA 1.

(i) *If* $e,M \xrightarrow{k} \underline{n},M'$ *then* $M' = M$ *and* $\langle S,M,e\cdot C\rangle \overset{*}{\Rightarrow} \langle \underline{n}\cdot S,M',C\rangle$.

(ii) *If* $b,M \xrightarrow{k} \left\{ \begin{matrix} \underline{\mathrm{true}} \\ \underline{\mathrm{false}} \end{matrix} \right\},M'$ *then* $M' = M$ *and* $\langle S,M,b\cdot C\rangle \overset{*}{\Rightarrow} \langle \left\{ \begin{matrix} \underline{\mathrm{true}} \\ \underline{\mathrm{false}} \end{matrix} \right\}\cdot S,M'.C\rangle$.

(iii) *If* $p,M \xrightarrow{k} \underline{\mathrm{null}},M'$, *then* $\langle S,M,p\cdot C\rangle \overset{*}{\Rightarrow} \langle S,M',C\rangle$.

Proof. We shall omit the proofs of (i) and (ii) and prove (iii) by induction on k (parts (i) and (ii) are simpler).

Basis k = 0. In this case p = null, M = M' and use IV(⇒).

Step. Assume (iii) for all k' < k, and assume

(1) $\quad p,M \xrightarrow{k} \mathrm{null},M' .$

Argue by cases

(a) p is null. Impossible, since $\underline{\mathrm{null}},M \not\rightarrow$.

(b) p is $(x_i:=e)$. Then k = 1 and (1) must have been inferred by III (→), so $e,M \xrightarrow{*} \underline{n},M$ and $M' = M[\underline{n}/i]$. So we have

$$\begin{aligned} &\langle S,M,(x_i:=e)\cdot C\rangle \Rightarrow \langle \underline{i}\cdot S,M,e\cdot \underline{\mathrm{assign}}\cdot C\rangle \\ &\overset{*}{\Rightarrow} \langle \underline{n}\cdot \underline{i}\cdot S,M,\underline{\mathrm{assign}}\cdot C\rangle \quad \text{by Lemma 1(i)} \\ &\Rightarrow \langle S,M[\underline{n}/i]\rangle \qquad\qquad \text{by V (⇒)}, \end{aligned}$$

as required.

(c),(d) We leave the cases p is $(p_1;p_2)$ or p is if b then p_1 else p_2 as an exercise for the reader.

(e) p is (while b do p_1). Then we have

$$p,M \rightarrow \left\{ \begin{matrix} p_1;p \\ \underline{\mathrm{null}} \end{matrix} \right\},M \xrightarrow{k-1} \underline{\mathrm{null}},M', \text{ with resp. } b,M \xrightarrow{*} \left\{ \begin{matrix} \underline{\mathrm{true}} \\ \underline{\mathrm{false}} \end{matrix} \right\},M$$

and Lemma 1(ii) then allows us to deduce

$$\begin{aligned}
\langle S,M,p\cdot C\rangle &\Rightarrow \langle p_1\cdot b\cdot S,M,b\cdot \underline{\text{while}}\cdot C\rangle && \text{by IV } (\Rightarrow)\\
&\Rightarrow \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\}\cdot p_1\cdot b\cdot S,M,\underline{\text{while}}\cdot C\rangle && \text{by Lemma 1(ii)}\\
&\Rightarrow \langle S,M,\left\{\begin{matrix}p_1;p\\ \underline{\text{null}}\end{matrix}\right\}\cdot C\rangle && \text{by V } (\Rightarrow)\\
&\overset{*}{\Rightarrow} \langle S,M',C\rangle && \text{by the ind.hypoth. at } k-1.
\end{aligned}$$

This concludes the proof of Lemma 1. □

Lemma 1 is half of our equivalence theorem. For the other half we introduce a definition.

DEFINITION. The reduction $\langle S,M,t\cdot C\rangle \overset{k}{\rightarrow} \langle S',M',C\rangle$ is *perfect* if the control stack in each intermediate state is a proper extension of C; that is, C is first "uncovered" at the last step.

LEMMA 2.

(i) *If* $\langle S,M,e\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = \underline{n}\cdot S$ *for some* $\underline{n}$, $M' = M$ *and* $e,M \overset{*}{\rightarrow} \underline{n},M$.

(ii) *If* $\langle S,M,b\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = \underline{\text{true}}\cdot S$ *or* $S' = \underline{\text{false}}\cdot S$, $M' = M$ *and* $b,M \overset{*}{\rightarrow} \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},M$ *resp.*

(iii) *If* $\langle S,M,p\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = S$ *and* $p,M \overset{*}{\Rightarrow} \underline{\text{null}},M'$.

Proof. Again, we omit the proofs of (i) and (ii), and deal with representative parts of (iii), for which we induce on k.

Basis k = 0. Impossible.

Step. Assume (iii) for all $k' < k$, and that

(2) $\langle S,M,p\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$

is perfect.

Argue by cases:

(a) p is $\underline{\text{null}}$. Then k must be 1 since the reduction (2) is perfect, and by IV (⇒) we see that M' = M, S' = S. The rest is trivial.

(b) p is $(x_i := e)$. Then from (2)

$$\left.\begin{array}{l} \langle S,M,p\cdot C\rangle \Rightarrow \langle \underline{i}\cdot S,M,e\cdot \underline{assign}\cdot C\rangle \\ \qquad \overset{k-2}{\Rightarrow} \langle S'',M'',\underline{assign}\cdot C\rangle \end{array}\right] \text{ perfect}$$

$$\Rightarrow \langle S',M',C\rangle \qquad \text{by V } (\Rightarrow),$$

where by Lemma 2(i), $S'' = \underline{n}\cdot S$, $M'' = M$ and $e,M \overset{*}{\to} \underline{n},M$. Hence by V $(\Rightarrow)$ $S'' = S$, $M' = [\underline{n}/i]M$; so $p,M \to \underline{null},M'$ by III $(\to)$.

(c),(d) p is $(p_1;p_2)$ or $(\underline{if}\ b\ \underline{then}\ p_1\ \underline{else}\ p_2)$. Exercise.

(e) p is $(\underline{while}\ b\ \underline{do}\ p_1)$. Then from (2)

$$\left.\begin{array}{l} \langle S,M,p\cdot C\rangle \Rightarrow \langle p\cdot b\cdot S,M,b\cdot \underline{while}\cdot C\rangle \\ \qquad \overset{k_1}{\Rightarrow} \langle S'',M'',\underline{while}\cdot C\rangle \end{array}\right] \text{ perfect}$$

(3) $$= \langle \left\{\begin{matrix}\underline{true}\\ \underline{false}\end{matrix}\right\} \cdot p\cdot b\cdot S,M,\underline{while}\cdot C\rangle \text{ where } b,M \overset{*}{\to} \left\{\begin{matrix}\underline{true}\\ \underline{false}\end{matrix}\right\},M \text{ resp.}$$

$$\Rightarrow \langle S,M,\left\{\begin{matrix}(p_1;p)\\ \underline{null}\end{matrix}\right\}\cdot C\rangle \text{ by V } (\Rightarrow)$$

$$\overset{k_2}{\Rightarrow} \langle S',M',C\rangle \qquad \text{perfectly, by assumption.}$$

Now $k_2 < k$, so using Lemma 2(iii) at k_2, for the last reduction, we infer that resp.

(4) $$\left\{\begin{matrix}(p_1;p)\\ \underline{null}\end{matrix}\right\},M \overset{*}{\to} \underline{null},M'.$$

But from (3), by VI $(\to)$ we obtain respectively

$$p,M \to \left\{\begin{matrix}p_1;p\\ \underline{null}\end{matrix}\right\},M,$$

whence $p,M \overset{*}{\to} \underline{null},M'$ by (4).

This completes the proof of Lemma 2. □

THEOREM. EVAL = eval.

Proof. We need to establish that

$$\langle \varepsilon, M, p \cdot \varepsilon \rangle \overset{*}{\Rightarrow} \langle \varepsilon, M', \varepsilon \rangle \quad \text{iff} \quad p', M \overset{*}{\rightarrow} \underline{\text{null}}, M'.$$

($\Rightarrow$) Suppose $\langle \varepsilon, M, p \cdot \varepsilon \rangle \overset{*}{\Rightarrow} \langle \varepsilon, M', \varepsilon \rangle$. Thus must be a perfect reduction, since there is no production of the form

$$\langle S, M, \varepsilon \rangle \Rightarrow \ldots .$$

So Lemma 2 provides the rest.
($\Leftarrow$) This is a simple application of Lemma 1. □

3. DENOTATIONAL SEMANTICS

3.1. Semantic domains

In this section we give a semantic description of L in terms of cpo's, using the definitions and results of the first two sections of "Models of LCF" [28]. But first we need two further cpo-preserving domain operations (we have already seen that if D and E are cpos, so is the continuous function domain [D→E], which we shall abbreviate henceforward to just D → E).

Cartesian product. If we define the ordering $\sqsubseteq$ over

$$D \times E = \{\langle x,y \rangle \mid x \in D,\ y \in E\}$$

by

$$\langle x,y \rangle \sqsubseteq \langle x',y' \rangle \text{ iff } x \sqsubseteq x' \text{ and } y \sqsubseteq y',$$

the following are easily verified:

(i) $D \times E$ is a cpo; indeed, $\perp_{D\times E} = \langle \perp_D, \perp_E \rangle$ and for the chain $\{\langle x_i, y_i \rangle \mid i \geq 0\}$ in $D \times E$, $\bigsqcup_i \langle x_i, y_i \rangle = \langle \bigsqcup_i x_i, \bigsqcup_i y_i \rangle$.

(ii) The pairing function $\lambda x \cdot \lambda y \cdot \langle x,y \rangle \in D \rightarrow E \rightarrow D \times E$ is continuous, and so are the selector functions

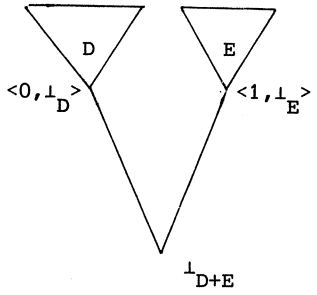
$$\text{fst} = \lambda\langle x,y\rangle \cdot x \in D \times E \to D$$
$$\text{snd} = \lambda\langle x,y\rangle \cdot y \in D \times E \to E.$$

Disjoint sum. Let us define

$$D + E = \{\bot\} \cup \{\langle 0,x\rangle \mid x \in D\} \cup \{\langle 1,x\rangle \mid x \in E\}$$

and the ordering $\sqsubseteq$ over $D + E$ by $z \sqsubseteq z'$ iff either $z = \bot$, or $z = \langle 0,x\rangle$ and $z' = \langle 0,x'\rangle$ and $x \sqsubseteq x'$ in D, or $z = \langle 1,y\rangle$ and $z' = \langle 1,y'\rangle$ and $y \sqsubseteq y'$ in E.

A diagram makes it clear:



The flags 0 and 1 are merely for disjoining D from E, so that for example $D + D$ contains two distinct copies of D.

Associated with + are the following functions:

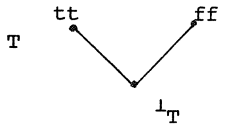
(i) The *injection* functions $\iota_0: D \to D + E = \lambda x{:}D \cdot \langle 0,x\rangle$
$\iota_1: D \to D + E = \lambda y{:}E \cdot \langle 1,y\rangle$.

(ii) The *projection* functions $\pi_0: D + E \to D$
$\pi_1: D + E \to E$,

where $\pi_0 z = \begin{cases} x \text{ if } z = \langle 0,x\rangle \\ \bot_D \text{ if } z = \langle 1,y\rangle \text{ or } \bot_{D+E} \end{cases}$,

and $\pi_1 z$ is similar.

(iii) *The discriminator functions* $\left.\begin{matrix}\delta_0: \\ \delta_1:\end{matrix}\right\} D + E \to T$



where $\delta_0 z = \begin{cases} \text{tt if } z = \langle 0,x\rangle \\ \text{ff} \quad " \quad = \langle 1,y\rangle \\ \bot_T \quad " \quad = \bot_{D+E} \end{cases}$,

and $\delta_1 z$ is similar with tt, ff interchanged.

It is a simple exercise to show that these are all continuous functions, and that $\forall x \in D$

(a) $\pi_0(\iota_0 x) = x$,

(b) $\pi_1(\iota_0 x) = \perp_E$,

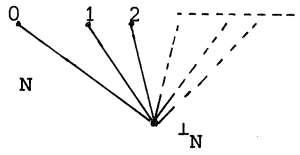
(c) $\delta_0(\iota_0 x) = tt$,

and many similar identities. Of course the binary + can be generalised to n-ary and even infinitary +. If for the latter we choose the non-negative integers 0,1,... as flags, we may also define

$$D^* = \sum_{i=0}^{\infty} D^i = \{\cdot\} + D + (D \times D) + (D \times D \times D) + \ldots$$

(where $\{\cdot\}$ is the one element domain), which is a domain of finite sequences of elements of D. Then $\iota_0(\cdot)$ is just the null sequence, and

$$\text{length: } D^* \to N$$



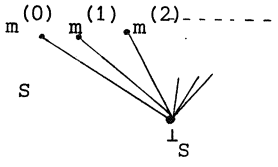
is just the "flag selecting" function. It is easy enough to define all the normal list processing operations - head, tail, cons, null - in terms of pairing and selecting and the ι_i, δ_i, π_i.

3.2. Denotation of language L

Our aim is to acribe directly (rather than via a machine) a function: $S \to S$ to each program in L, as its meaning or denotation, where S is now a cpo of (abstract) memories or stores. We choose S to respect convention: it is accidental that S stood also for a stack in our operational semantics, but we shall henceforward use it only in the new sense.

Though there are alternatives, we choose S to be a flat cpo of vectors of non-negative integers. More precisely

$$S = \{\perp_S\} \cup \{\langle m_i \rangle \mid \langle \underline{m}_i \rangle \in \text{Memories}\}$$



and we shall allow s to vary over S, but will adopt the convention that m varies over $S - \{\perp_S\}$ - i.e. it stands always for a *defined* store; also we will use m for the abstract counterpart of M:

$$M = \underline{m}^\bullet, \underline{m}_1, \ldots \iff m = m_0, m_1, \ldots \quad .$$

The only operations we need on stores are

$$\text{update: } N \to (N \times S \to S)$$
$$\text{select: } N \to (S \to N) \ ,$$

where

$$\text{update } i(n,s) = \begin{cases} \perp_S \text{ if } i,n \text{ or } s \text{ is undefined} \\ [n/i]s \qquad \text{otherwise} \end{cases} ,$$

$$\text{select } i\ s \quad = \begin{cases} \perp_N \text{ if } i \text{ or } s \text{ is undefined} \\ s_i \qquad \text{otherwise.} \end{cases}$$

These functions are easily shown continuous, and we can also shown

LEMMA 1. *If* i,j,n_1,n_2 *are all defined then*

(1) $$\text{select } i \text{ (update } j(n,m)) = \begin{cases} n, \text{ if } i = j \\ \text{select } i\ m \text{ if } i \neq j \ , \end{cases}$$

(2) $$\text{update } i \text{ (select } i\ m, m) = m \ ,$$

(3) $$\text{update } i \ (n_1, \text{update } j(n_2, m)) = \begin{cases} \text{update } i\ (n_1, m) \text{ if } i = j \\ \text{update } j\ (n_2, \text{update } i\ (n_1, m)) \text{ otherwise.} \end{cases}$$

Proof. Omitted. □

REMARK. Part (3) holds even for undefined i,j,n_1,n_2 and for $\perp_S$ in place of m.

3.3. Semantics of expressions

We first assign to each expression e a function $\in S \to N$ as its meaning, and to each boolean expression b a function $\in S \to T$. The meaning $E[\![e]\!]$ of e

is given as follows:

$$\begin{aligned}
&\mathcal{E}[\![e]\!]\perp_S &&= \perp_N\\
&\mathcal{E}[\![\underline{n}]\!]m &&= n\\
&\mathcal{E}[\![x_i]\!]m &&= \text{select } i\ m\ (= m_i)\\
&\mathcal{E}[\![e_1+e_2]\!]m &&= \mathcal{E}[\![e_1]\!]m + \mathcal{E}[\![e_2]\!]m\\
&\ldots \text{ etc. for all iops.}
\end{aligned}$$

[Note that the right-hand + stands for a function $\in N \to (N\to N)$, while the left-hand + is a symbol of L.]

The brackets $[\![\ ]\!]$ are to distinguish syntactic objects. For later work we shall need to consider a cpo E, which consists of all Expressions together with certain "partially-defined" expressions; then it will be possible to discuss $\mathcal{E}$ as a member of the domain $E \to (S\to N)$. Such a domain as E would indeed be important if we were discussing expressions as data objects (on a par with N) - as they would be for a compiler for example. But here we need do no more than remark that $\mathcal{E}[\![e]\!]$ is defined inductively on the structure of e. The same remark applies to our later semantic functions $\mathcal{B}$ and $\mathcal{P}$.

For boolean expressions, we define $\mathcal{B}[\![b]\!] \in S \to T$ thus:

$$\begin{aligned}
&\mathcal{B}[\![b]\!]\perp_S &&= \perp_S\\
&\mathcal{B}[\![\underline{\text{true}}]\!]m &&= \text{tt},\quad \mathcal{B}[\![\underline{\text{false}}]\!]m = \text{ff}\\
&\mathcal{B}[\![e_1=e_2]\!]m &&= \begin{cases} \perp_T & \text{if either } \mathcal{E}[\![e_1]\!]m \text{ or } \mathcal{E}[\![e_2]\!]m \text{ is } \perp_N\\ \text{tt} & \text{if they are equal}\\ \text{ff} & \text{otherwise.}\end{cases}
\end{aligned}$$

Before dealing with programs, we state without proof a simple lemma which relates the denotational semantics of expressions to their operational semantics.

LEMMA 2.

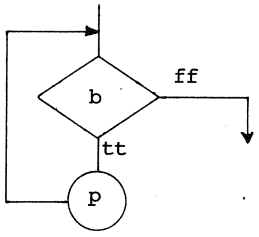
(1) $e,M \overset{*}{\to} \underline{n},M$ iff $\mathcal{E}[\![e]\!]m = n \neq \perp_N$

(2) $b,M \overset{*}{\to} \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},M$ iff $\mathcal{B}[\![b]\!]m = \left\{\begin{matrix}\text{tt}\\ \text{ff}\end{matrix}\right\}$.

Proof. By induction on the structure of expressions. Recall our convention that m is the abstract counterpart of M. □

3.4. Semantics of programs

We proceed as with expressions to define $\mathcal{P}[\![p]\!] \in S \to S$ inductively on the structure of programs. But first the while construct deserves special attention. What function $f \in S \to S$ is $\mathcal{P}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1]\!]$? The diagram



suggests that f satisfies

$$fs = \mathcal{B}[\![b]\!]s \to f(\mathcal{P}[\![p_1]\!]s), s$$

so that we choose for f the least fixed point fix Φ of the functional

$$\Phi = \lambda f \cdot \lambda s \cdot (\mathcal{B}[\![b]\!]s \to f(\mathcal{P}[\![p_1]\!]s), s) \in [[S \to S] \to [S \to S]].$$

That the *least* fixed point of Φ is right will be justified by our theorem.

In defining $\mathcal{P}[\![e]\!]$ we choose not to *define* $\mathcal{P}[\![e]\!]\perp_S = \perp$ since it *follows* from our definition as an easy lemma.

$$\begin{aligned}
&\mathcal{P}[\![\underline{\text{null}}]\!]s && = s\\
&\mathcal{P}[\![x_i := e]\!]s && = \text{update } i\ (\mathcal{E}[\![e]\!]s, s)\\
&\mathcal{P}[\![p_1 ; p_2]\!]s && = \mathcal{P}[\![p_2]\!](\mathcal{P}[\![p_1]\!]s)\\
&\mathcal{P}[\![\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2]\!]s && = \mathcal{B}[\![b]\!]s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s\\
&\mathcal{P}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1]\!] && = \text{fix}(\lambda f \cdot \lambda s' \cdot \mathcal{B}[\![b]\!]s' \to f(\mathcal{P}[\![p]\!]s'), s').
\end{aligned}$$

LEMMA 3. $\mathcal{P}[\![p]\!]\perp_S = \perp_{S'}$.

Proof. By induction on the structure of p. Use the definition of update, and of $\mathcal{B}[\![b]\!]$, and also that fix $\Phi s = \Phi(\text{fix } \Phi)s$. □

3.5. Equivalence of operational and denotational semantics for L

We now prove the theorem:

THEOREM. $P[\![p]\!]m = m'$ *iff* $p,M \overset{*}{\to} \underline{null},M'$.

As an easy corollary of this, we have that $P[\![p]\!]m = \perp_S$ iff the reduction of p,M under $\to$ fails to terminate. It is easier to divide the theorem into two lemmas.

LEMMA 4. *If* $p,M \to p',M'$ *then* $P[\![p]\!]m = P[\![p']\!]m'$.

Proof. Induction on p.

Basis. (i) p is null. Nothing to prove, since $null,M \not\to$.

(ii) p is $(x_i:=e)$. Then by the rules of $\to$, p' is null, and $M' = [\underline{n}/i]M$, where $e,M \overset{*}{\to} \underline{n},M$. So Lemma 2 gives $E[\![e]\!]m = n$, whence $P[\![p]\!]m = \text{update } i\ (n,m) = m[n/i] = m'$, while $P[\![p']\!]m' = P[\![\underline{null}]\!]m' = m'$.

Step. (iii) p is $(p_1;p_2)$.
If p_1 *is* null, then $p',M' = p_2,M$ by rule IV ($\to$). But then $P[\![p]\!]m = P[\![p_2]\!](P[\![\underline{null}]\!]m) = P[\![p_2]\!]m = P[\![p']\!]m'$.
Otherwise $p' = (p_1';p_2)$ by IV ($\to$), where $p_1,M \to p_1',M'$, so by Induction Hypothesis $P[\![p_1]\!]m = P[\![p_1']\!]m'$. But then $P[\![p]\!]m = P[\![p_2]\!](P[\![p_1]\!]m) = P[\![p_2]\!](P[\![p_1']\!]m') = P[\![p']\!]m'$.

(iv) p is (if b then p_1 else p_2). Exercise.

(v) p is (while b do p_1). Then

$$p',M' = \left\{\begin{matrix} p_1;p \\ \underline{null} \end{matrix}\right\},M \text{ where resp. } b,M \overset{*}{\to} \left\{\begin{matrix} \underline{true} \\ \underline{false} \end{matrix}\right\},M.$$

So Lemma 2 gives resp. $B[\![b]\!]m = \left\{\begin{matrix} tt \\ ff \end{matrix}\right\}$, whence

$$\begin{aligned} P[\![p]\!]m &= (\text{fix } \Phi)m = \Phi(\text{fix } \Phi)m \\ &= B[\![b]\!]m \to (\text{fix } \Phi)(P[\![p_1]\!]m),m \\ &= \text{resp. } \left\{\begin{matrix} \text{fix } \Phi(P[\![p_1]\!]m) = P[\![p]\!](P[\![p_1]\!]m) \\ m = P[\![\underline{null}]\!]m \end{matrix}\right\} \\ &= \text{in each case } P[\![p']\!]m'. \quad \square \end{aligned}$$

Since this lemma was only concerned with one step reductions, we needed no induction to handle the while construct.

LEMMA 5. *If* $\mathcal{P}[\![p]\!]m = m'$ *then* $p,M \overset{*}{\to} \underline{null},M'$.

Proof. Again, by induction on p. We leave all the cases to the reader as an exercise, except for the while construct. Here, we assume the lemma for p_1 and assume $\mathcal{P}[\![p]\!]m = m'$, where p is (while b do p_1). Now $\mathcal{P}[\![p]\!] = \text{fix } \Phi$ (Φ as before). Let

$$\left.\begin{aligned} f_0 &= \bot_{s\to s} \\ f_{i+1} &= \Phi\ f_i \end{aligned}\right\} \text{ so that } \mathcal{P}[\![p]\!] = \bigsqcup_i f_i \ .$$

We shall prove by induction on i (i.e. induction on the *iterates* of Φ) that

(#) $\qquad$ if $f_i m = m'$ then $p,M \overset{*}{\to} null,M'$.

Further, from $\mathcal{P}[\![p]\!]m = \bigsqcup_i (f_i m) = m'$ we can infer that for *some* i, $f_i m = m'$; the rest follows from (#).
[Note: this inference is a consequence of the fact that S is a *flat* cpo; $\langle f_i m\rangle$ is a chain in S with lub m', and hence some member of the chain is itself equal to m'.]

Proof of (#). For the basis i = 0 there is nothing to prove, since $f_0 m = \bot m = \bot$, while m' is by convention *defined*.

Step. Assume # for i, and assume $f_{i+1} m = m'$. Now

$$f_{i+1} m = \Phi f_i m = (\mathcal{B}[\![b]\!]m \to f_i(\mathcal{P}[\![p_1]\!]m), m) = m'.$$

But since m' is defined

either a) $\mathcal{B}[\![b]\!]m = tt$ (whence $b,M \overset{*}{\to} \underline{true},M$ by Lemma 2) and
$\qquad f_{i+1} m = f_i(\mathcal{P}[\![p_1]\!]m) = m'$,
or $\quad$ b) $\mathcal{B}[\![b]\!]m = ff$ (whence $b,M \overset{*}{\to} \underline{false},M$) and
$\qquad f_{i+1} m = m = m'$.

In the case of b), $p,M \overset{*}{\to} \underline{null},M'$ follows easily. For case a), we first

note that $P[\![p_1]\!]m$ must be defined, = m" say. (If not, then we have $m' = f_i(\perp) \sqsubseteq P[\![p]\!]\perp = \perp$, a contradiction.)

So by the inductive assumption of Lemma 5 for p_1, we have that $p_1,M \overset{*}{\to} \underline{null},M"$; and by the present inductive assumption for f_i we have that $p,M" \overset{*}{\to} \underline{null},M'$. It follows that

$$\begin{aligned} p,M &\to (p_1;p),M \quad (\text{since } b,M \overset{*}{\to} \underline{true},M) \\ &\overset{*}{\to} (\underline{null},p),M" \\ &\to p,M" \\ &\overset{*}{\to} \underline{null},M' \end{aligned}$$

as required.

Proof of Theorem. (⇒) Directly from Lemma 5.
(⇐) Let $p,M = p^{(0)},M^{(0)} \to \ldots \to p^{(n)},M^{(n)} = \underline{null},M'$. Then Lemma 4 tells us that $\{P[\![p^{(i)}]\!]m^{(i)}\}$ are all equal, whence $P[\![p]\!]m = P[\![\underline{null}]\!]m' = m'$ as required. □

4. CONTINUATION SEMANTICS

4.1. Continuations

Hitherto we have tacitly assumed of the language L that the execution of each program construct, if it terminates at all, terminates "naturally" - i.e. control passes out at the end of the construct. Thus it was safe to write

$$P[\![p_1;p_s]\!]s = P[\![p_2]\!](P[\![p_1]\!]s)$$

indicating that p_2 will always be executed after the termination of p_1.

Various language constructs do not admit this assumption; jumps are the obvious example, and error exits (trapped or not) are another. We shall consider the latter only, and show how the device of *continuations* - introduced independently by L. Morris and C. Wadsworth - can handle abnormal exits. Extension of the technique to jumps involves no further concepts - it is just rather tedious, essentially because jumps spoil the structured

nature of programs.

To avoid too much detail, let us from now on forget the details of assignments (or other basic non-compound instructions) and expressions in L, merely assuming that there are certain Boolean expressions $b_1, b_2, \ldots$ with corresponding semantics $\beta_i = \mathcal{B}[\![b_i]\!] \in S \to T$, and certain basic instructions c_i - including assignments (but excluding null) with corresponding semantics $\gamma_i = C[\![c_i]\!] \in S \to S$, where in particular $C[\![x_i := e]\!]s = \text{update } i\ (E[\![e]\!]s, s)$. (We are thus preventing consideration of abnormal exit from expressions or basic instructions; the technique of continuations adapts easily to allow this.)

Consider now adding a single extra instruction "error", whose effect is supposed to be to abort the whole program and deliver an error message (which may depend on the current store). There is no easy way to fit error into the semantic equations for $\mathcal{P}$.

So we proceed as follows. First, assume a cpo O containing all possible end results of programs, including error messages. We should need something like O anyway - hitherto we have taken the meaning of a program in the domain $S \to S$, but we are not often interested in the state of the whole store at the end of the whole program. With O we may imagine extracting the final output by applying to $\mathcal{P}[\![p]\!]s$ some output function $\in S \to O$.

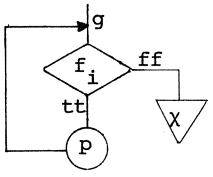
(Aside: we are still not considering programs which can output - or input - information during execution; it turns out that continuations also make this easy to handle.)

But now it appears that if χ is such an output function, then $\chi \circ \mathcal{P}[\![p]\!]$ is a function in the same domain $S \to O$, respecting the work done by executing p and then "outputting". If we call $C = S \to O$ the domain of *continuations*, we can equally well specify the meaning of a program p by defining the effect of "prefixing" its work to our arbitrary continuation χ to yield another continuation. So our new semantic function will be $\mathcal{Q}$, and we define $\mathcal{Q}[\![p]\!] \in C \to C$ below.

To emphasize the back-to-front way of working, consider the while construct again. If p is while b_i do p_1, then we want to specify the *output* $\mathcal{Q}[\![p]\!]\chi s$ which results from starting p with s and continuing after p with χ.

Let $\mathcal{Q}[\![p]\!] = g$. Then $g\chi$ satisfies

$$g\chi s = \beta_i s \to \mathcal{Q}[\![p_1]\!](g\chi)s,\ \chi s$$



and our equation below settles on the least fixed point of this recursive equation for g.

$$\begin{aligned}
&\mathcal{Q}[\![\underline{\text{null}}]\!]\chi s &&= \chi s\\
&\mathcal{Q}[\![c_i]\!]\chi s &&= \chi(\gamma_i s)\\
&\mathcal{Q}[\![p_1;p_2]\!]\chi s &&= \mathcal{Q}[\![p_1]\!](\mathcal{Q}[\![p_2]\!]\chi)s\\
&\mathcal{Q}[\![\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2]\!]\chi s &&= \beta_i s \rightarrow \mathcal{Q}[\![p_1]\!]\chi s, \mathcal{Q}[\![p_2]\!]\chi s\\
&\mathcal{Q}[\![\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1]\!] &&= \text{fix}(\lambda g\cdot\lambda\chi\cdot\lambda s\cdot\beta_i s \rightarrow \mathcal{Q}[\![p_1]\!](f\chi)s, \chi s),
\end{aligned}$$

and finally we add

$$\mathcal{Q}[\![\underline{\text{error}}]\!]\chi s = \text{dump } s\ ,$$

where dump is a special error continuation - we may imagine that it prints the whole store if we like. The vital point is that because error chooses to ignore the normal continuation χ, any program containing error may also choose to ignore its normal continuation. (Write out the meaning of the program "error;p" to emphasize this, and also notice the difference between $\mathcal{P}[\![p_1;p_2]\!]$ and $\mathcal{Q}[\![p_1;p_2]\!]$.)

In these lectures we do not propose to develop the semantics of more complex languages (they can be found in the literature) but rather to look at proofs about simple languages; in fact we shall only do *one* proof, since we also want to examine the possibility of mechanizing it. But before leaving errors, it is worth while sketching an extension to the language and its semantics to allow for trapping errors.

Suppose then that error is to invoke not a fixed continuation "dump", but an error continuation η which has somehow been established by the program. That is, to give the meaning of error, our new semantic function $\mathcal{R}$ needs an η as well as a χ as argument - i.e. $\mathcal{R}[\![p]\!] \in C \rightarrow C \rightarrow C$, and we write

$$\mathcal{R}[\![\underline{\text{error}}]\!]\chi\eta s = \eta s \qquad \text{(cannot succeed!)}$$

and naturally

$$\mathcal{R}[\![\underline{\text{null}}]\!]\chi\eta s = \chi s \qquad \text{(cannot fail!)} \quad .$$

But how are errors to be trapped - or (to ask the same question differently) how are error continuations established? The simplest possible answer is to add the program construct

$$p_1 \ \underline{\text{orelse}} \ p_2 \ ,$$

whose effect is to be as p_1 if p_1 terminates normally, otherwise to execute p_2 as soon as p_1 commits an error. (Thus, the whole construct can only err if p_2 errs.)

Exercise. Give the semantic equation

$$\mathcal{R}[\![p_1 \ \underline{\text{orelse}} \ p_2]\!] \chi \eta s = \ldots .$$

and complete the definitions of $\mathcal{R}$. You should then be able to *prove*

(i) orelse is associative, i.e.

$$\mathcal{R}[\![p_1 \ \underline{\text{orelse}} (p_2 \ \underline{\text{orelse}} \ p_3)]\!] = \mathcal{R}[\![(p_1 \ \underline{\text{orelse}} \ p_2) \underline{\text{orelse}} \ p_3]\!] \ .$$

(ii) error is a left zero for " ; ", i.e.

$$\mathcal{R}[\![\underline{\text{error}}; p]\!] = \mathcal{R}[\![\underline{\text{error}}]\!] \ .$$

(Is it a right zero?)

(iii) error is a left identity for orelse, i.e.

$$\mathcal{R}[\![\underline{\text{error}} \ \underline{\text{orelse}} \ p]\!] = \mathcal{R}[\![p]\!] \ .$$

(Is it a right identity? Is there a left or right zero for orelse?)

Does orelse distribute over " ; " ? ... over if b_i then -- else -- ?
If not always, then under what conditions?

Of course the most useful errors are those which return some kind of value, but we shall have to omit this kind of extension. Again, the techniques require no really new idea.

4.2. Techniques for proof about continuous functions

Most interesting properties of language semantics involve fixed-points, and their proofs depend upon the fact that fix Φ denotes the *least* fixed point of Φ. (The simple properties of error and orelse mentioned above are an exception.) The fundamental method is to prove that the required property holds (or something similar holds) when fix Φ is replaced by each of the iterates $f_i = \Phi^i(\perp)$ of Φ. Let us take as an example a general property of fixed points:

$$F(\mathrm{fix}(G\circ F)) = \mathrm{fix}(F\circ G),$$

provided F,G are continuous.

Proof. Let

$$H = F\circ G, \quad J = G\circ F$$

and let

$$h_i = H^i(\bot), \quad j_i = J^i(\bot).$$

We then show by induction on i that

(*) $$F(j_i) \sqsubseteq \mathrm{fix}(F\circ G),$$

whence

$$F(\mathrm{fix}\ J) = F(\bigsqcup_i j_i) = \bigsqcup_j (F(j_i)) \sqsubseteq \mathrm{fix}(F\circ G),$$

and a similar induction on the iterates h_i yields the other half of the required result.

To prove (*):

when $i = 0$, $F(j_0) = F(\bot) \sqsubseteq F(G(\mathrm{fix}\ H)) = (F\circ G)(\mathrm{fix}(F\circ G)) = \mathrm{fix}(F\circ G)$;

otherwise $F(j_{i+1}) = F((G\circ F)j_i) = (F\circ G)(Fj_i) \sqsubseteq (F\circ G)(\mathrm{fix}(F\circ G))$ (by induction)

$$= \mathrm{fix}(F\circ G).$$

Hence the induction is complete. □

Exercises. Prove similarly

(i) $\mathrm{fix}\ F = \mathrm{fix}(F\circ F)$;

(ii) if $F(\bot) = G(\bot) = \bot$ and $F\circ G = G\circ F$ then $\mathrm{fix}\ F = \mathrm{fix}\ G$;

(iii) ditto, replacting the second condition by $F\circ F\circ G = G\circ F$.

The method of such proofs is to prove first that $\bar{F}[f_i]$ holds for all i, and then step to $\bar{F}[\mathrm{fix}\ \Phi]$ where $f_i = \Phi^i\bot$. For this step to be valid, the predicate $\bar{F}[\ ]$ has to be *directed-complete*; that is, for *any* chain $\langle f_i \rangle$,

$$(\forall_i \cdot \bar{F}[f_i]) \Rightarrow \bar{F}[\bigsqcup_i f_i].$$

Now any equational formula - that is t = t' - or inequality $t \sqsubseteq t'$ is easily shown to be directed - complete considered as a predicate of some free variable x in the formula, provided that t and t' are built by application and abstraction from continuous functions. (This is the import of Prop. 3.1 in "Models of LCF".) It is also easy to show that if $F[x]$ is directed complete in x, so are

$$\forall y \cdot F[x]$$

$$G \Rightarrow F[x] \ ,$$

provided that x is not a free variable of G. The class of directed-complete formulae may be extended further; we merely emphasize here the importance of the notion.

4.3. The equivalence of direct and continuation semantics

We have presented two semantic definitions of language L, both guided by our intuition about what L *should* mean, and one of them (the *direct* semantics P) further substantiated by a proof of its equivalence with an operational definition. We must therefore answer the question: *in what sense do P and Q give the same meaning to* L? In some sense they simulate one another - their definitions are structurally similar - but we cannot simply claim $P = Q$, or $P[\![p]\!] = Q[\![p]\!]$, since the domains are different.

The simulation relation between direct and continuation semantics for a more complex language was exhibited by Reynolds. He used more powerful techniques than we have developed here, but for L they are unnecessary.

We will give a rather simple proof of the appropriate relationship between P and Q, and then proceed to examine how the proof might be formalized and performed interactively with a machine.

We must first omit from L the error command, since it was not handled by P. Then in view of our discussion when Q was first defined, it is natural to expect that

$$(1) \qquad \forall \chi \cdot Q[\![p]\!]\chi = \chi \circ P[\![p]\!]$$

and indeed this is readily verified from the semantic equations when p is c_i or null.

But suppose that p_0 is in some program like

$$\underline{\text{while}}\ \underline{\text{true}}\ \underline{\text{do}}\ p_1$$

which never terminates? It is not hard to verify that whatever χ, $\mathcal{Q}[\![p_0]\!]\chi = \perp$; on the other hand if we pick $\chi = \lambda s \cdot o$ for some $o \neq \perp \in O$ (i.e. χ is a constant function) then we also have $\chi \circ \mathcal{P}[\![p]\!] = \lambda s \cdot o \neq \perp$.

So some restriction on equation (1) is required. It is not hard to accept that χ should be a *strict* function - that is, it should satisfy $\chi\perp = \perp$; intuitively the continuation should be patient enough to wait for p to introduce some information (which in our case means a fully defined store). We therefore formulate our simulation relation thus

$$(2) \qquad \forall p \cdot \forall \chi \cdot \chi \text{ strict} \Rightarrow \mathcal{Q}[\![p]\!]\chi = \chi \circ \mathcal{P}[\![p]\!] \ ,$$

which we shall prove inductively on the structure of programs. We also need to assume that the meanings of basic instructions and boolean expressions are strict functions:

$$(3) \qquad \forall i \ \beta_i(\perp_S) = \perp_T, \quad \gamma_i(\perp_S) = \perp_S \ .$$

The following Lemma is needed:

LEMMA. $\forall p \cdot \forall \chi \cdot \chi$ strict $\Rightarrow \mathcal{Q}[\![p]\!]\chi$ strict.

Proof. By induction on the structure of p. Assume χ strict.

Basis. (i) $p = \underline{\text{null}}$. Then $(\mathcal{Q}[\![p]\!]\chi)\perp = \chi\perp = \perp$.

(ii) $p = c_i$. Then $(\mathcal{Q}[\![p]\!]\chi)\perp = \chi(\gamma_i\perp) = \chi\perp$ by (3)
$= \perp$.

Step. Assume the lemma for all subprograms of p

(iii) $p = (p_1;p_2)$. Then $\mathcal{Q}[\![p]\!]\chi = \mathcal{Q}[\![p_1]\!]\chi'$ where $\chi' = \mathcal{Q}[\![p_2]\!]\chi$. But by the Lemma for p_2 χ' is strict, hence by the lemma for p_1 so is $\mathcal{Q}[\![p]\!]\chi$.

(iv) $p = (\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)$. Then $\mathcal{Q}[\![p]\!]\chi\perp = \beta_i\perp \rightarrow \ldots\ , \ldots = \perp$ by (3).

(v) $p = (\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1)$. Then $\mathcal{Q}[\![p]\!]\chi\perp = (\text{fix}\ \Phi)\chi\perp$ $(\Phi = \lambda g \cdot \lambda\chi \cdot \lambda s \cdot \beta_i s \rightarrow \ldots\ ,$
$\ldots) = \Phi(\text{fix}\ \Phi)\chi\perp = \beta_i\perp \rightarrow \ldots\ , \ldots = \perp$ by (3). □

REMARK. In this proof the while construct caused no difficulty because it executes the test before the body. You may like to try the following exercise, in which you may need an inner induction on the iterates of a functional like Φ.

Exercise. Formulate the obvious continuation semantics of the construct

$$\underline{\text{do}}\ p_1\ \underline{\text{until}}\ b_i$$

and prove the corresponding case of the lemma. Also prove that

$$\mathcal{Q}[\![\underline{\text{if}}\ b_i\ \underline{\text{then}}\ (\underline{\text{do}}\ p_1\ \underline{\text{until}}\ b_i)\ \underline{\text{else}}\ \underline{\text{null}}]\!]$$

$$= \mathcal{Q}[\![\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1]\!] .$$

(For the last part you may need to look at the style of proof of the following Theorem.)

SIMULATION THEOREM. $\forall p \cdot \forall \chi \cdot \chi$ strict $\Rightarrow \mathcal{Q}[\![p]\!]\chi = \chi \circ \mathcal{P}[\![p]\!]$.

Proof. By induction on the structure of p.

Basis. (i) $p = \underline{\text{null}}$, (ii) $p = c_i$. Both trivial.

Step. Assume the theorem for all subprograms of p, and assume χ strict.

(iii) $p = (p_1;p_2)$. Then $\mathcal{Q}[\![p]\!]\chi = \mathcal{Q}[\![p_1]\!]\chi'$ where $\chi' = \mathcal{Q}[\![p_2]\!]\chi$
$= \chi' \circ \mathcal{P}[\![p_1]\!]$ by the theorem for p_1, since χ' is strict by the Lemma,
$= \chi \circ \mathcal{P}[\![p_2]\!] \circ \mathcal{P}[\![p_1]\!]$ by the theorem for p_2
$= \chi \circ \mathcal{P}[\![p]\!]$.

(iv) $p = (\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)$. Then $\mathcal{Q}[\![p]\!]\chi s = \beta_i s \to \mathcal{Q}[\![p_1]\!]\chi s, \mathcal{Q}[\![p_2]\!]\chi s$
while $(\chi \circ \mathcal{P}[\![p]\!])s = \chi(\beta_i s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s)$.
The result follows by considering the three cases $\beta_i s = tt, ff, \perp_T$.

(v) $p = (\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1)$. Then $\mathcal{Q}[\![p]\!]\chi = (\text{fix}\ \Psi)\chi$ and $\chi \circ \mathcal{P}[\![p]\!] = \chi \circ \text{fix}\ \Phi$
where $\Psi = \lambda g \cdot \lambda \chi' \cdot \lambda s' \cdot \beta_i s' \to \mathcal{Q}[\![p_1]\!](g\chi')s', \chi' s'$
$\Phi = \lambda f \cdot \lambda s' \cdot \beta_i s' \to (f \circ \mathcal{P}[\![p_1]\!])s', s'$.

We need only show that if f_i, g_i are the iterates of Φ, Ψ respectively, then

for each $i \geq 0$

(4) $\quad \forall \chi' \cdot \chi' \text{strict} \Rightarrow g_i\chi' = \chi' \circ f_i$.

For then, when χ is strict $\mathcal{Q}[\![p]\!]\chi = (\bigsqcup_i g_i)\chi = \bigsqcup_i (g_i\chi) = \bigsqcup_i (\chi \circ f_i) = \chi \circ \bigsqcup_i f_i = \chi \circ \mathcal{P}[\![p]\!]$.

To prove (4); when $i = 0$ $g_0\chi' = \bot\chi' = \bot = \chi' \circ \bot$ (χ' strict) $= \chi' \circ f_0$. Now assume (4) for i, and let χ' be strict. Then

$$g_{i+1}\chi' = \beta_i s \to \mathcal{Q}[\![p_1]\!](g_i\chi')s, \chi's$$

$$= \beta_i s \to (g_i\chi' \circ \mathcal{P}[\![p_1]\!])s, \chi's \text{ by the theorem for } p_1, \text{ since}$$
$$g_i\chi' \sqsubseteq (\text{fix } \Psi)\chi' = \mathcal{Q}[\![p]\!]\chi' \text{ which is strict}$$
$$\text{by the lemma, so } g_i\chi' \text{ is also strict,}$$

$$= \beta_i s \to (\chi' \circ f_i \circ \mathcal{P}[\![p_1]\!])s, \chi's \text{ by inductive assumption for } i.$$

On the other hand

$$(\chi' \circ f_{i+1})s = \chi'(f_{i+1}s) = \chi'(\beta_i s \to (f_i \circ \mathcal{P}[\![p_1]\!])s, s)$$

and equality follows by case analysis on $\beta_i s$. □

5. MECHANIZED SEMANTICS

5.1. Deductive systems

The aim of this section is to explore the possibility of mechanizing the proof of the simulation theorem, using a formal deductive calculus which is an extension of that described in "Models of LCF", Sections 3 & 4. The extension is in two directions; more logical connectives, and more types. This system has been presented in detail in Milner, Morris and Newey [24], and we shall be more informal about it here.

Well-formed formulae (wffs). Wffs are formed from the atomic wffs (awffs) by normal use of the connectives &, ⇒, ∀ (conjunction, implication and universal quantification), and a sentence is $\Gamma \vdash A$, where Γ is a set of

wffs and A is a wff. As rules of inference we add

Conjunction $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$ *Selection* $\dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A \quad \Gamma \vdash B}$

Deduction $\dfrac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \Rightarrow B}$ *Modus Ponens* $\dfrac{\Gamma \vdash A \quad \Delta \vdash A \Rightarrow B}{\Gamma \cup \Delta \vdash B}$

Generalization $\dfrac{\Gamma \vdash A}{\Gamma \vdash \forall x \cdot A}$ *Specialization* $\dfrac{\Gamma \vdash \forall x \cdot A}{\Gamma \vdash A\{t/x\}}$

(x not free in Γ)

Assumption $\dfrac{}{\{A\} \vdash A}$

[Note: these rules actually replace INCL, CONT and CUT of "Models of LCF".]

Types. Instead of just two basic types IND and TR, we allow any number of basic types (including TR which we rename T), and types may be built using the binary connectives + × and →. From what has gone before, we know that there is a domain (a cpo) for each type, whose name is just that type.

Two further ingredients are necessary; (a) Reflexive types, and (b) Polymorphic types. From Scott, we know that any family of recursive domain equations

$$D_1 = F_1(D_1,\ldots,D_n)$$

$$D_n = F_n(D_1,\ldots,D_n)$$

has a solution for the D_i (strictly the equality is an isomorphism) where the F_i are built from the D_i, and possibly other domains, by + × and →.

We therefore allow any family of such equations as relations over our type constants (domain names); more precisely then a type is any equivalence class of type expressions induced by these relations, and it may be named by any member of the class. These are our reflexive types.

But there are many operations which make sense at an infinity of types. Examples are the conditional and fixed point operations, and the operations fst, snd, δ_0, δ_1, ι_0, ι_1, π_0, π_1 introduced earlier. To allow

these operations to have types, we introduce type variables $\alpha,\alpha_1,\ldots$, $\beta,\beta_1,\ldots$ and then for example fst: $\alpha\times\beta \to \alpha$, ι_0: $\alpha \to \alpha+\beta$ etc.

Simple and natural rules apply for a term or wff to be well-typed, and we do not go into them here. Type variables have no binding quantifier, but we add to our deduction rules the following

Type Instantiation $\dfrac{\Gamma \vdash A}{\Gamma \vdash A\{\tau/\alpha\}}$ where α is a type variable not in Γ, and τ is any type (possibly including variables).

This rule is invoked whenever we wish to use a "polymorphic" theorem, such as

$$\forall x \cdot \pi_0(\iota_0 x) \equiv x$$

at a particular instance of its type.

This is the basis which is intended to formalize our informal reasoning within a typed framework; from now on we shall omit types almost everywhere (except when discussing them, rather than the objects which possess them), and we would expect any tolerable mechanization to allow these omissions but to supply and check types internally.

Note: We will use "fix" rather than "Y" for the fixed point combinator. Also we shall use the constant " $\perp$ " - instead of "UU" as in "Models of LCF" - to denote $\perp$. It may be worth remarking here that the reasons for choosing TT, FF, UU in LCF were (i) t, f are far too often used as variables; similarly with T,F. (ii) In addition, the Stanford LCF was superimposed on LISP, which does quite surprising things with the atom T. (iii) TT and FF are much quicker to type than true, false. However, I suggest we pronounce TT,FF,UU "true", "false", "bottom".

5.2. Formalizing the syntax of language L

Our calculus discusses cpo's; so to discuss both the syntax *and* the semantics of L, the syntactic objects as well as the semantic ones must be found in suitable cpo's. (This will mean introducing things like the undefined program, and possibly infinite programs; these do not get in the way however!)

We define a set of mutually reflexive types

(1)
$$\begin{aligned} \mathrm{PROGM} &= \mathrm{NULL} + \mathrm{INSTN} + \mathrm{COMPD} + \mathrm{CONDL} + \mathrm{ITERN} \\ \mathrm{NULL} &= \cdot \\ \mathrm{INSTN} &= \ldots \\ \mathrm{COMPD} &= \mathrm{PROGM} \times \mathrm{PROGM} \\ \mathrm{CONDL} &= \mathrm{BEXP} \times \mathrm{PROGM} \times \mathrm{PROGM} \\ \mathrm{ITERN} &= \mathrm{BEXP} \times \mathrm{PROGM} \\ \mathrm{BEXP} &= \ldots \end{aligned}$$

Assume that + and × associate to the right.

" · " is our name for the domain with a single element, which we shall denote by the constant (). This domain is exiomatized easily by

$$\vdash \forall x_0 \cdot x_0 \equiv () \qquad !$$

We have left out the definitions of INSTN and BEXP, consistent with our previous treatment. They could indeed by left unspecified, and for our theorem all we shall need is that the functions

$$\mathcal{C}\colon \mathrm{INSTN} \to S \to S \quad \text{and} \quad \mathcal{B}\colon \mathrm{BEXP} \to S \to T$$

satisfy

(2)
$$\vdash \forall c_{\mathrm{INSTN}} \cdot \mathcal{C}[\![c]\!]\perp_S \equiv \perp_S$$
$$\vdash \forall b_{\mathrm{BEXP}} \cdot \mathcal{B}[\![b]\!]\perp_S \equiv \perp_T$$

Note that our type equations really give the *abstract* syntax of L; all that is said of a while program is that it is a pair consisting of a boolean expression and a program, which is all that matters to us.

But we would like mnemonic names for the discriminators, constructors and destructors of this abstract syntax; we *define*

Discriminators
$$\begin{aligned} &\vdash \mathrm{isnull} \equiv \delta_0 \\ &\vdash \mathrm{isinstn} \equiv \lambda p \cdot \delta_1 p \to \delta_0(\pi_1 p), FF \\ &\vdash \mathrm{iscompd} \equiv \lambda p \cdot \delta_1 p \to (\delta_1 \pi_1 p \to \delta_0(\pi_1(\pi_1 p)), FF), FF \\ &\quad \text{etc.} \end{aligned}$$

all of type PRGM → T .

Constructors (3)

$$\vdash \text{mknull} \equiv \iota_0 : \text{NULL} \to \text{PROGM}$$
$$\vdash \text{mkinstn} \equiv \iota_1 \circ \iota_0 : \text{INSTN} \to \text{PROGM}$$
$$\vdash \text{mkcompd} \equiv \iota_1 \circ \iota_1 \circ \iota_0 : \text{PROGM} \times \text{PROGM} \to \text{PROGM}$$
etc.

Destructors

$$\vdash \text{destnull} \equiv \pi_0 : \text{PROGM} \to \text{NULL}$$
$$\vdash \text{destinstn} \equiv \pi_0 \circ \pi_1 : \text{PROGM} \to \text{INSTN}$$
$$\vdash \text{destcompd} \equiv \pi_0 \circ \pi_1 \circ \pi_1 : \text{PROGM} \to \text{PROGM} \times \text{PROGM}$$
etc.

We have worked with a *binary* disjoint sum operation (rather than a 5-*ary* one) which makes these definitions lengthy. But once we have proved standard theorems for our new operations, their definitions need never be seen again. Such theorems are

(4)

$$\vdash \text{isnull}(\text{mknull}()) \equiv \text{TT}$$
$$\vdash \forall b \forall p_1 \forall p_2 \cdot \text{iscondl}(\text{mkcondl}(b, p_1, p_2)) \equiv \text{TT}$$
$$\vdash \forall b \forall p_1 \forall p_2 \cdot \text{destcondl}(\text{mkcondl}(b, p_1, p_2)) \equiv \text{TT}$$
$$\vdash \forall i \cdot \text{isnull}(\text{mkinstn}(i)) \equiv \text{FF}$$
etc., etc..

We shall use them later as simplification rules (which we shall describe) in proving the main theorem - indeed, these theorems themselves are proved by using the earlier definitions as simplification rules, i.e. they are proved completely automatically.

Structural Induction for L

We wish to derive, from the standard rule of computation induction given in "Models of LCF", the following inference rule for programs of L:

(5)

$$\frac{\begin{array}{lll} \vdash F[\bot] & \vdash F[\text{mknull}()] & \vdash F[\text{mkinstn}(i)] \\ F[p_1], F[p_2] \vdash F[\text{mkcompd}(p_1, p_2)] & & \\ F[p_1], F[p_2] \vdash F[\text{mkcondl}(b, p_1, p_2)] & & \\ F[p_1] \vdash F[\text{mkitern}(b, p_1)] & & \end{array}}{\vdash F[p]} ,$$

where extra assumptions on the left of the turnstiles have been left out

for clarity, but may occur (with their union occurring in the conclusion of the rule) provided they contain none of i,b,p_1,p_2,p free.

We now have to face a problem of all reflexive domain equations, that they do not necessarily have a unique solution. Our rule will only follow if the domains PROGM,... are in some sense the *least* solution of equations (1). The following axiom ensures this; what it expresses is roughly that every program is well-founded - i.e. if we analyse it into its primitive components (in NULL, INSTN and BEXPN) and then build it up again, we have back our original program. The axiom is made more concise with the following functional #:

$$\vdash \forall f \forall g \forall x \forall y \cdot (f\#g)(x,y) \equiv (f(x),g(y)).$$

The axiom is:

(6) $\quad \vdash \lambda p \cdot p \equiv \text{fix progfun},$

where

$$\begin{aligned} \vdash \text{progfun} \equiv \lambda f \cdot \lambda p \cdot\ & \text{isnull } p \rightarrow \text{mknull(destnull } p), \\ & \text{isinstn } p \rightarrow \text{mkinstn(destinstn } p), \\ & \text{iscompd } p \rightarrow \text{mkcompd((f\#f)(destcompd } p)), \\ & \text{iscondl } p \rightarrow \text{mkcondl(((}\lambda b \cdot b)\#(f\#f))\text{(destcondl } p)), \\ & \text{isitern } p \rightarrow \text{mkitern(((}\lambda b \cdot b)\#f)\text{(destitern } p)), \\ & \perp. \end{aligned}$$

Now we are at last ready to derive the structural induction rule (5), for an arbitrary formula $F[p]$.

We take the instance of the computational induction rule on the functional progfun, in which $G[f]$ is $\forall p \cdot F[f(p)]$:

$$(7) \qquad \frac{\vdash G[\perp] \quad G[f] \vdash G[\text{progfun}(f)]}{\vdash G[\text{fix}(\text{progfun})]} \ .$$

Our task is to prove the two hypotheses of (7) from the six hypotheses of (5); (7) then allows us to infer (together with (6)) that $\vdash G[\lambda p \cdot p]$, i.e. $\vdash \forall p \cdot F[p]$, whence the conclusion of (5) follows by specialization.

Basis. $G[\bot]$ is $\forall p \cdot F[\bot(p)]$, or $\forall p \cdot F[\bot]$ which is the generalization of the first hypothesis of (5).

Step. Assume

(8) $G[f]$, that is $\forall p \cdot F[f(p)]$.

We require to prove $\forall p \cdot F[\text{progfun}(f)p]$, so we attempt to prove $F[\text{progfun}(f)p]$, for some arbitrary p.
For this, it is enough to consider the truth values of the five conditions isnull(p),isinstn(p),... . Now from the definition of progfun, the only cases which do not yield progfun(f)p $\equiv \bot$ (when we are done) are when some condition is TT and the earlier ones are all FF. In each case, progfun(f)p is equal to one of the five expressions at the right of $\rightarrow$ in (6). Consider just the fourth case. Then we are trying to prove

$$F[\text{mkcondl}(((\lambda b \cdot b)\#(f\#f))(\text{destcondl}(p)))]$$

that is (for some b,p_1 and p_2)

$$F[\text{mkcondl}(b,f(p_1),f(p_2))],$$

which follows readily, by (8), from the fifth hypothesis of (5). □

5.3. Strategies

We now attempt to describe the kind of proof strategy which can relieve one of a morass of technical detail in performing proofs interactively with a machine, using the Simulation Theorem as an example. Because induction (either structural or computational) appears to be the major creative ingredient in such proofs, there is some hope that without too much ill-directed search the machine can automatically dispose of large parts of a proof, given only an initial hint of what induction to perform. Indeed, once this is achieved one may expect to design strategies which make an intelligent search for the right induction; such strategies have already been studied with some success (though in restricted problem domains) by Boyer and Moore [25], Aubin [26] and von Henke [27] among others.

A major part of such strategies will be the use of equational formulae as *simplification rules* (which we abbreviate to *simprules*). More

precisely, any theorem of the form

$$\Gamma \vdash \forall x_1,\ldots,x_n \cdot t[x_1,\ldots,x_n] \equiv t'[x_1,\ldots,x_n],$$

which belongs to the current *simplification set* (*simpset*) will be used in the following way when simplification is explicitly called for in a strategy: to transform a *goal* F (formula to be proved) into a simpler goal F', any subterm having the form $t[u_1,\ldots,u_n]$ is replace by $t'[u_1,\ldots,u_n]$. If after all possible such replacements F' is discovered to be a simple tautology, then the goal is achieved. As the strategy proceeds, transforming the original goal into a list of subgoals, each subgoal attains a simpset appropriate for its proof.

Other components of a suitable strategy for our example will emerge in the following analysis; the resulting strategy will be seen to be not especially oriented to the example, though we would certainly not claim universal applicability for it.

As a starting point, let us now specify our main goal $G_0[p]$ as follows:

$$(G_0) \qquad \forall \chi \cdot \chi\bot \equiv \bot \Rightarrow \forall s \cdot \mathcal{Q}[\![p]\!]\chi s \equiv \chi(\mathcal{P}[\![p]\!]s)$$

and suppose that we have in the initial simpset the following theorems:

1. Each clause of the definitions of $\mathcal{P}$ and $\mathcal{Q}$, in the form*)
   $\vdash \forall b,p_1,p_2,s \cdot \mathcal{P}[\![\text{mkcondl}(b,p_1,p_2)]\!]s \equiv \mathcal{B}[\![b]\!]s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s$, together with the strictness clauses $\mathcal{P}[\![\bot_{\text{PROGM}}]\!] = \bot$, $\mathcal{Q}[\![\bot_{\text{PROGM}}]\!] = \bot$.
2. All the theorems (4) above concerning the syntax of L.
3. The strictness theorems (2) for $\mathcal{B}$ and $\mathcal{C}$.
4. Standard rules such as β conversion, conditional conversion ($\vdash \forall x,y \cdot TT \to x,y \equiv x$, etc.), minimality ($\forall x \cdot \bot x \equiv x$) and rules concerning fst, snd, δ_i, π_i, ι_i.

---

*) although we continue to use [\![ ]\!] as decorative brackets, they have no formal significance to distinguish them from ( ).

First tactic

Try structural induction on p, then simpify the resulting six subgoals; for each remaining subgoal add its assumptions to the simpset for that goal.

Applying this to G_0 gives (before simplification) the six hypotheses of rule (5) (with G_0 for F). Simplification however eliminates the first three - those with no assumptions - provided a wide enough class of simple tautologies is detected. The first hypothesis, for example, becomes

$$\forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot\bot \equiv \chi\bot$$

and the other two yield even simpler tautologies. We are therefore left with

(G_1) $\forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot\mathcal{Q}[\![p_1]\!](\mathcal{Q}[\![p_2]\!]\chi)s \equiv \chi(\mathcal{P}[\![p_2]\!](\mathcal{P}[\![p_1]\!]s))$

with $G_0[p_1]$ and $G_0[p_2]$ in the simpset,

(G_2) $\forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot\mathcal{B}[\![b]\!]s \to \mathcal{Q}[\![p_1]\!]\chi s, \mathcal{Q}[\![p_2]\!]\chi s \equiv \chi(\mathcal{B}[\![b]\!]s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s)$

with $G_0[p_1]$ and $G_0[p_2]$ in the simpset,

(G_3) $\forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot \mathrm{fix}\,\Psi\,\chi s \equiv \chi(\mathrm{fix}\,\Phi\,s)$ *)

with $G_0[p_1]$ in the simpset.

But the reader will have already noticed that the various formulae $G_0[p_1]$ added to the simpsets are not equational, and we must therefore extend our notion of simprule to justify what we have done. We introduce the notion of a *conditional* simprule. Any theorem of the form

$$\Gamma \vdash \forall\underline{y}(t_2[\underline{y}] \equiv t_2'[\underline{y}] \Rightarrow \forall x\cdot t_1[\underline{x},\underline{y}] \equiv t_1'[\underline{x},\underline{y}])$$

(in which $\underline{y}$, $\underline{x}$ stand for vectors of variables) may be used in simplifying a goal F as follows: any subterm of F having the form $t_1[\underline{u},\underline{v}]$ may be

---

*) We use Ψ and Φ here as *abbreviations* for our familiar functionals; they are not variables of the calculus.

replaced by $t_1'[\underline{u},\underline{v}]$ (where $\underline{u},\underline{v}$ are vectors of terms) *provided that*

(*) $\quad t_2[\underline{v}] \equiv t_2'[\underline{v}]$

is first proved by simplification. For pragmatic purposes we add one further constraint: that each of the terms v_i should be *free in* F, i.e. no variable occurrence free in v_i is bound in F. This constraint is not applied to the u_i, either here or in ordinary simplification; it is present to prevent conditional simplification setting up for itself too many unachievable subsidiary goals of the form of (*) above. We will see how it works later. [Aside: even our ordinary simplification mechanism (without conditional simprules) runs the resk of non-termination, unless some constraint is placed on the *form* of simplification rules. Without this, perhaps it should be called *computation* rather than simplification; the point is that some automatic equational transformation is needed, and it can always be bounded artificially in some way.]

With this extended notion, we may now assume that our strictness lemma

$$\vdash \forall p \cdot \chi \cdot \chi\perp \equiv \perp \Rightarrow Q[\![p]\!]\chi\perp \equiv \perp$$

is present in the simpset throughout.

Second tactic

Each of G_i (i = 1,2,3) is a quantified implication, and we now iterate the process of *stripping* quantifiers and *assuming* antecedents. These are normal informal proof techniques, and are justified respectively by the rules of generalisation and deduction given earlier. We choose arbitrary new variables for those which become unbound.

We then add the assumed antecedent into the simpset for each subgoal, and apply simplification.

What happens to G_1 under this tactic? Before the simplification it becomes

$$Q[\![p_1]\!](Q[\![p_2]\!]\chi_1)s_1 \equiv \chi_1(P[\![p_2]\!](P[\![p_1]\!]s_1))$$

with $G_0[p_1]$, $G_0[p_2]$ and $\chi_1\perp \equiv \perp$ in the simpset.

Now the subterm $\mathcal{Q}[\![p_2]\!]\chi_1$, being *free*, is an admissible instance for χ in the conditional simprule $G_0[p_1]$, enabling the left hand side to become

(*) $\qquad \mathcal{Q}[\![p_2]\!]\chi_1(\mathcal{P}[\![p_1]\!]s),$

provided that

$$\mathcal{Q}[\![p_2]\!]\chi_1\perp \equiv \perp$$

can be proved by simplification. But this is done by use of the strictness lemma, and by $\chi_1\perp \equiv \perp$. Returning to (*), it is similarly transformed further (using $G_0[p_2]$) into the right hand side.

Thus G_1 has become a trivial equation, and is achieved.

What happens to G_2 under the second tactic? Similar use of conditional simplification easily reduces it (as the reader may like to check) to

$$\mathcal{B}[\![b]\!]s_2 \to \chi_2(\mathcal{P}[\![p_1]\!]s_2),\chi_2(\mathcal{P}[\![p_2]\!]s_2) \equiv \chi_2(\mathcal{B}[\![b]\!]s_2 \to \mathcal{P}[\![p_1]\!]s_2,\mathcal{P}[\![p_2]\!]s_2)$$

(G_4) with $G_0[p_1]$, $G_0[p_2]$ and $\chi_2\perp \equiv \perp$ in the simpset.

The simple task of proving G_4 we leave to the third tactic.

What happens to G_3 under the second tactic? Before simplification it becomes

$$\mathrm{fix}(\lambda g\cdot\lambda\chi'\cdot\lambda s'\cdot(\mathcal{B}[\![b]\!]s' \to \mathcal{Q}[\![p_1]\!](g\chi')s',\chi's'))\chi_3 s_3$$

(G_5) $\qquad \equiv \chi_3(\mathrm{fix}(\lambda f\cdot\lambda s'\cdot\mathcal{B}[\![b]\!]s' \to f(\mathcal{P}[\![p_1]\!]s'),s'))s_3)$

with $G_0[p_1]$ and $\chi_3\perp \equiv \perp$.

Now in this case, conditional simplification by $G_0[p_1]$ is not possible, since the required instance $g\chi'$ of χ in that rule is not free in G_5. (If we relaxed this constraint on conditional simplification, because χ' is bound in the left hand side of G_5 we would need to prove $\chi'\perp \equiv \perp$ for an *arbitrary* χ' to achieve the subsidiary goal of the conditional simplification, and this is not valid.)

So simplification does nothing for G_5.

Third tactic

A tactic which will achieve G_4 and is of wide application is: find any term t of type T which is free in a goal G, and produce three subgoals, each consisting of G with respectively t ≡ TT, t ≡ FF and t ≡ ⊥ in the simpset. Then simplify.

G_4 yields very simply to this tactic - the only candidate for t is $\mathcal{B}[\![b]\!]s_2$ - and the strictness of χ_2 disposes of the third subgoal. On the other hand G_5 is not amenable to the tactic, since $\mathcal{B}[\![b]\!]s'$ is not free in G_5.

The third tactic is of wider applicability than to the type T; one may perform case analysis on any term denoting a member of a finite domain.

5.4. Discussion

Let us suppose that our overall strategy is to apply the three tactics in sequence, each to all the subgoals remaining after applying the previous one. The effect for our example is to reduce the original goal G_0 to a single subgoal G_5, for which as we saw earlier a further induction is required. Let us briefly discuss G_5, then consider strategies in general.

The form of G_5 is

$$(\text{fix } \Psi)\chi_3 s_3 \equiv \chi_3((\text{fix } \Phi)s_3),$$

with $G_0[p_1]$ and $\chi_3\perp \equiv \perp$ in the simpset. Now our informal proof consisted in an inductive proof that for each i ≥ 0

$$(*) \qquad \forall\chi\cdot\chi\perp \equiv \perp \Rightarrow \forall s\cdot g_i\chi s \equiv \chi(f_i s),$$

where g_i, f_i are the iterates of Ψ, Φ. Formally we might therefore apply the rule of *parallel* induction:

$$\frac{F[\perp,\perp] \quad F[f,g] \vdash F[\Phi f,\Psi g]}{[F \text{ fix } \Phi, \text{fix } \Psi]}$$

(which is easily derived from the standard rule), taking for F[f,g] the formula (*) with suffix i removed. (In passing we may remark that the

principle reason for using such induction rules rather than *mathematical* induction on the index of the iterates is to avoid formalizing arithmetic solely for this purpose.)

Without troubling with details, we claim that our original strategy, but with structural induction replaced by parallel induction will indeed achieve the goal G_3 (from which G_5 came); but it is worth remarking that the generation of G_5, resulting from uniform application of all three tactics, was wasted work.

The detailed study of this example leaves us with the impression that strategies for certain classes of problems may often be built from rather general purpose tactical material, but that it would be unwise to pursue the ideal of a single general purpose strategy.

For this very reason, a user of an interactive proof system must have the ability to extend not only his repertoire of Theorems, but also his repertoire of tactics, of ways of composing strategies from them, and hence of strategies themselves. This sounds very like programming; the problem then is to give him a programming language (a meta-LCF) in which it can all be done tolerably, and in which however badly he programs he cannot "prove" non-theorems. He will then not need to study a strategy at length in the abstract before typing it in and trying it; if he thinks that

> *structural induction on p, then stripping quantifiers and antecedents and doing case analysis, all mixed up with simplification*

is a recipe worth trying, then he can type it in, at perhaps no greater length than the above sentence, and see what happens.

Work with LCF at Stanford [18,19,20,21,22,23] has motivated the design of such a programming meta-language, and at Edinburgh we have implemented one which appears to allow plenty of scope for experiment in strategy-building.

As for the deductive calculus, we would not claim that the one outlined in this paper is the final answer, even for problems in the rather special area of programming language semantics; we expect to continue to find problems whose expression or solution is either impossible or repulsive however much we enrich the calculus. But in contrast to this, it *is* reasonable to expect that many pragmatic aspects of interactive proof-finding remain constant as the calculus varies. That is to say, a

good *meta*-language for proof may not be so far away. (This project has involved four people - initially L. Morris and M. Newey, and currently M. Gordon and C. Wadsworth - besides myself; much of the work remains still to be reported, but I would like to acknowledge here the very able and persistent work of these colleagues.)

6. LITERATURE

Scott and Strachey [1] give a starting point for the study of denotational semantics. Scott [2] provided the underlying models; in [3] he gives an outline without too much technical detail, and in [4] he carries the theory further.

Further studies in denotational semantics are given by Tennent [5], which contains both a good introduction and a presentation of the semantics of Reynolds' GEDANKEN. Mosses [6] presents ALGOL 60, and Gordon [7] presents LISP.

For operational semantics, Landin [10] gives a starting point. The description of PL/1 is by Lucas and Walk [11]; also Wegner [12] gives a very readable account of the Vienna Definition Language which was invented for the description of PL/1. Plotkin [13] at a more fundamental level discusses evaluation in the λ-calculus.

The continuation technique of Wadsworth and Morris is presented by Reynolds [14], who also gives a mathematical discussion of the directed complete relations which are employed in his analogue of our simulation theorem. Strachey and Wadsworth [15] illustrate the continuation technique.

For a study of the syntactic properties of formulae which express directed complete relations - i.e. formulae which admit the use of computation induction - see Klebansky et al [16] and Igarashi [17]. The forerunner of the computation induction was "recursion induction" given by McCarthy in [8]. Scott's rule was also discovered independently by Park [9].

The implementation of LCF carried out at Stanford in 1971-2 is described in Milner [18], and studies in its use are Milner and Weyhrauch [19], Weyhrauch and Milner [20], Newey [22], Aiello, Aiello and Weyhrauch [21] and von Henke [23]. The extended formal calculus used in the present paper is given in full detail in Milner, Morris and Newey [24].

Work on strategies for proof by induction can be found in Boyer and

Moore [25], Aubin [26] and von Henke [27]. The models of the original LCF are in Milner [28].

This is by no means a full list of the relevant papers, but should help the reader to explore further the different aspects of work in the field of semantics and proof.

REFERENCES.

[1] D. SCOTT & C. STRACHEY, *Towards a mathematical Semantics for Computer Languages,* Proc. Symposium on Computers and Automata, Microwave Res. Inst. Symposia series, Vol. 21, Polytechnic Institute of Brooklyn, 1971.

[2] D. SCOTT, *Lattice Theoretic Models for Various Type-free Calculi,* Proc. IV-th International Congress for Logic, Methodology and the Philosophy of Science, Bucharest, 1972.

[3] D. SCOTT, *Outline of a Mathematical Theory of Computation,* Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, 1970.

[4] D. SCOTT, *Data Types as Lattices,* Unpublished Lecture, University of Amsterdam, 1973.

[5] R. TENNENT, *The Denotational Semantics of Programming Languages,* Comm. A.C.M., Vol. 19, No. 8, 1976.

[6] P. MOSSES, *The Mathematical Semantics of ALGOL 60,* Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group, 1974.

[7] M. GORDON, *Models of pure LISP,* Experimental Programming Report 37, School of Artificial Intelligence, University of Edinburgh, 1973.

[8] J. McCARTHY, *A Basis for a Mathematical Theory of Computation,* Computer Programming and Formal System, D. Braffort & D. Hirshberg eds., North-Holland, Amsterdam, 1963.

[9] D. PARK, *Fixpoint Induction and Proofs of Program Properties,* Machine Intelligence 5, B. Meltzer & D. Michie eds., Edinburgh University Press, 1969.

[10] P. LANDIN, *The Mechanical Evaluation of Expressions*, Computer Journal 6, 4, 1964.

[11] P. LUCAS & K. WALK, *On the formal Descriptions of PL/]*, Annual Review in Automatic Programming 6, 3, 1969.

[12] P. WEGNER, *The Vienna Definition Language*, ACM Computing Surveys, 4, 1, 1972.

[13] G. PLOTKIN, *Call by name, call by value and the λ-calculus*, Theoretical Computer Science, 1, 2, 1975.

[14] J. REYNOLDS, *On the relation between direct and continuation semantics*, Proc. 2-nd Colloquium on Automata, Languages and Programming, Saarbrücken, 1974.

[15] C. STRACHEY & C. WADSWORTH, *Continuations: A mathematical semantics for handling full jumps*, Technical Monograph PRG-11, Oxford University, Computing Laboratory, Programming Research Group, 1974.

[16] B. KLEBANSKY, Z. MANNA & A. PNUELI, *Formulas admissible for Induction*, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1973.

[17] S. IGARASHI, *Admissibility of Fixed-Point Induction in First Order Logic of Typed Theories*, AI Memo AIM-168, Computer Science Dept., Stanford University, 1972.

[18] R. MILNER, *Logic for Computable Functions; Description of a Machine Implementation*, AI Memo No. 169, Computer Science Dept., Stanford, 1972.

[19] R. MILNER & R. WEHRAUCH, *Proving compiler correctness in a Mechanized Logic*, in Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

[20] R. WEYHRAUCH & R. MILNER, *Program semantics and correctness in a Mechanized Logic*, Proc. USA - Japan Computer Conference, Tokyo, 1972.

[21] L. AIELLO, M. AIELLO & R. WEYHRAUCH, *The semantics of PASCAL in LCF*, AIM-221 Computer Science Dept., Stanford University, 1974.

[22] M. NEWEY, *Formal Semantics of LISP with applications to program correctness*, AIM-243, Stanford University, Computer Science Dept., 1975.

[23] F. VON HENKE, *Notes on Automating theorem proving in LCF*, forthcoming memorandum, 1976.

[24] R. MILNER, L. MORRIS & M. NEWEY, *A Logic for Computable Functions with Reflexive and Polymorphic Types*, Proc. Conference on Proving and Improving Programs, Arc-et-Senans, 1975.

[25] R. BOYER & J. MOORE, *Proving Theorems about LISP functions*, J. ACM 22, 1975 (pp.129-144).

[26] R. AUBIN, *Some generalization heuristics in proofs by induction*, Proc. Conference on Proving and Improving Programs, Arc-et-Senans, 1975.

[27] F. VON HENKE, *On Automating Proofs by Induction*, unpublished paper, 1976.

[28] R. MILNER, *Models of LCF*, AI Memo No. 186, Computer Science Dept., 1973 (also in this volume; see p.49).